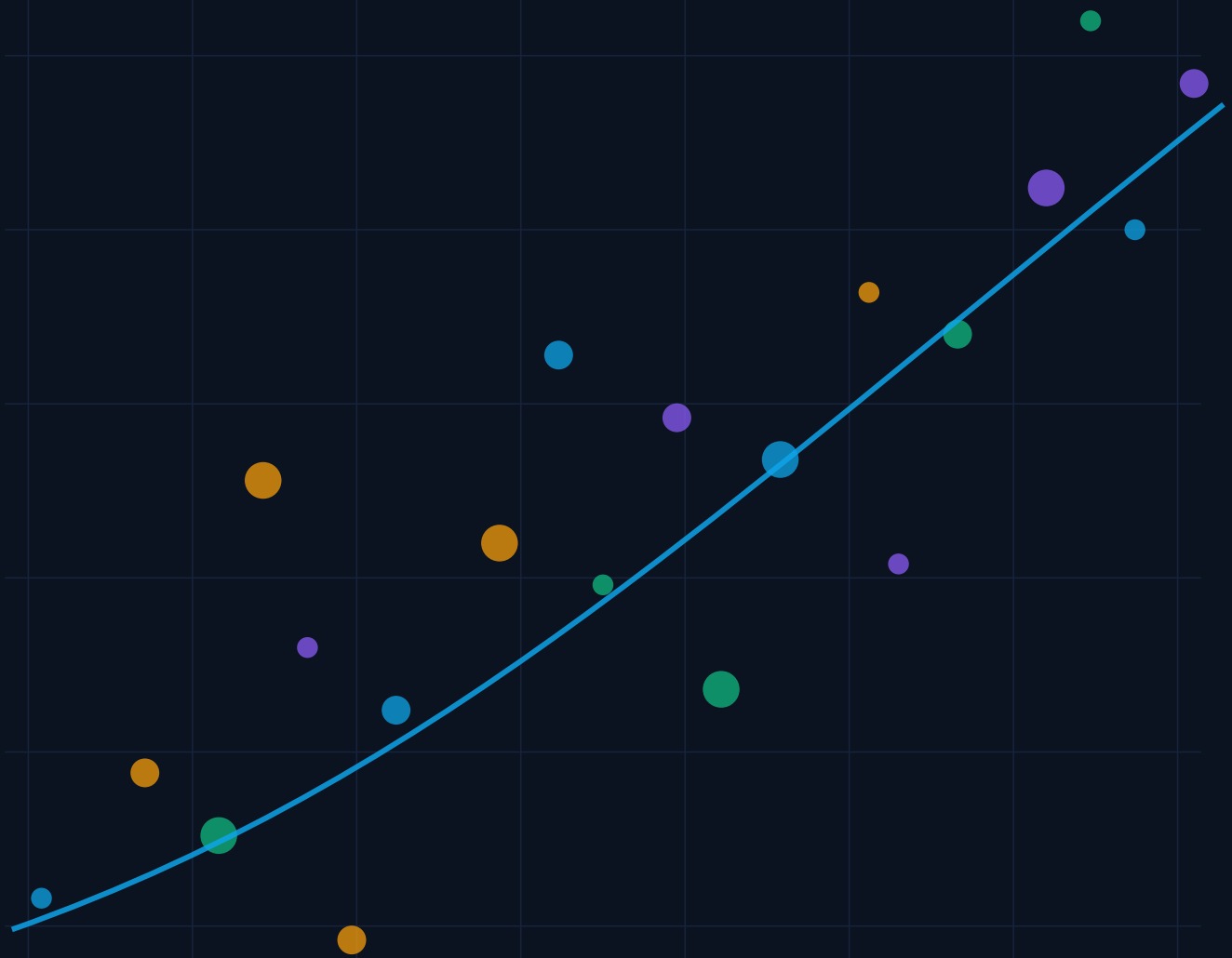
GUIA TÉCNICO · R

ggplot2

A Gramática dos Gráficos

Componha, entenda e customize qualquer gráfico —
e leve-o à web interativa com Shiny.



dados + aes + geom + stat + scale + coord + facet + theme

Julho de 2026

Sumário

A GRAMÁTICA DOS GRÁFICOS **3**

01 · Por que uma gramática? . . . 3

02 · Dados e aes(): o mapeamento estético . . . 5

03 · Geoms: as formas geométricas . . . 7

04 · Stats e position: o cálculo por trás do desenho . . . 9

05 · Scales: o tradutor entre dado e percepção . . . 11

06 · Coordenadas: a projeção final . . . 13

07 · Facetas: small multiples . . . 15

08 · Temas: customizando cada pixel não-dado . . . 16

09 · Estudo de caso: do padrão ao publication-ready . . . 18

SHINY: DA GRAMÁTICA À WEB INTERATIVA **20**

10 · Arquitetura: UI, server e o contrato entre eles . . . 20

11 · Reatividade a fundo: o grafo que trabalha por você . . . 22

12 · ggplot2 programável: o gráfico que o usuário monta . . . 24

13 · Performance e boas práticas . . . 26

14 · Cheat sheet: consulta rápida . . . 27

Como ler este guia. A Parte I constrói a gramática dos gráficos peça por peça — cada capítulo é um componente da equação da capa, com código executável e a figura que ele produz. A Parte II leva o mesmo gráfico à web com Shiny, do app mínimo à interação direta sobre os pontos. Todos os exemplos usam datasets embutidos (`mpg`, `diamonds`, `economics`): abra o RStudio e execute em paralelo — a gramática só se fixa compondo.

PARTE I

A GRAMÁTICA DOS GRÁFICOS

ggplot2 — composição, não catálogo de tipos

01 Por que uma gramática?

Ferramentas tradicionais tratam gráficos como um catálogo fechado: função para barras, função para dispersão, função para pizza. Cada uma com seus próprios argumentos, seus próprios limites. O ggplot2 (Wickham, 2010, sobre a *Grammar of Graphics* de Wilkinson, 2005) parte de outro princípio: **um gráfico é uma sentença composta por camadas independentes e ortogonais**. Assim como um conjunto finito de regras gramaticais gera infinitas frases, um conjunto finito de componentes gera qualquer gráfico — inclusive os que ainda não têm nome.

A equação fundamental que estrutura todo este guia:

Componente	Pergunta que responde	Exemplo
data	Quais dados? (formato tidy)	`mpg, diamonds`
aes()	Qual variável vira qual atributo visual?	`aes(x = displ, colour = drv)`
geom_*	Com que forma geométrica desenhar?	`geom_point(), geom_line()`
stat_*	Que transformação estatística aplicar antes?	`stat_count(), stat_smooth()`
position	Como resolver sobreposição?	`"stack", "dodge", "jitter"`
scale_*	Como traduzir valores em posições/cores/tamanhos?	`scale_y_log10()`
coord_*	Em que sistema de coordenadas projetar?	`coord_flip(), coord_polar()`
facet_*	Repetir em painéis por subgrupo?	`facet_wrap(~ class)`
theme()	Como estilizar tudo que não é dado?	`theme_minimal(), element_text()`

Na prática, você raramente escreve todos os nove: cada geom traz stat e position padrão, e cada aesthetic mapeada cria sua escala automaticamente. Customizar é **substituir o padrão do componente certo** — e é por isso que quem domina a gramática customiza qualquer gráfico: nunca há um argumento mágico a decorar, há sempre um componente a trocar.

```r
library(ggplot2)

# Um gráfico é uma SOMA de camadas, não um "tipo" fechado
ggplot(mpg, aes(x = displ, y = hwy)) +     # 1. dados + mapeamento estético
  geom_point() +                           # 2. camada geométrica
  geom_smooth(method = "loess") +          # 3. outra camada (stat + geom)
  scale_colour_manual(values = paleta) +   # 4. escalas
  facet_wrap(~ drv) +                      # 5. facetas
  coord_cartesian(ylim = c(10, 45)) +      # 6. coordenadas
  theme_minimal()                          # 7. tema
```

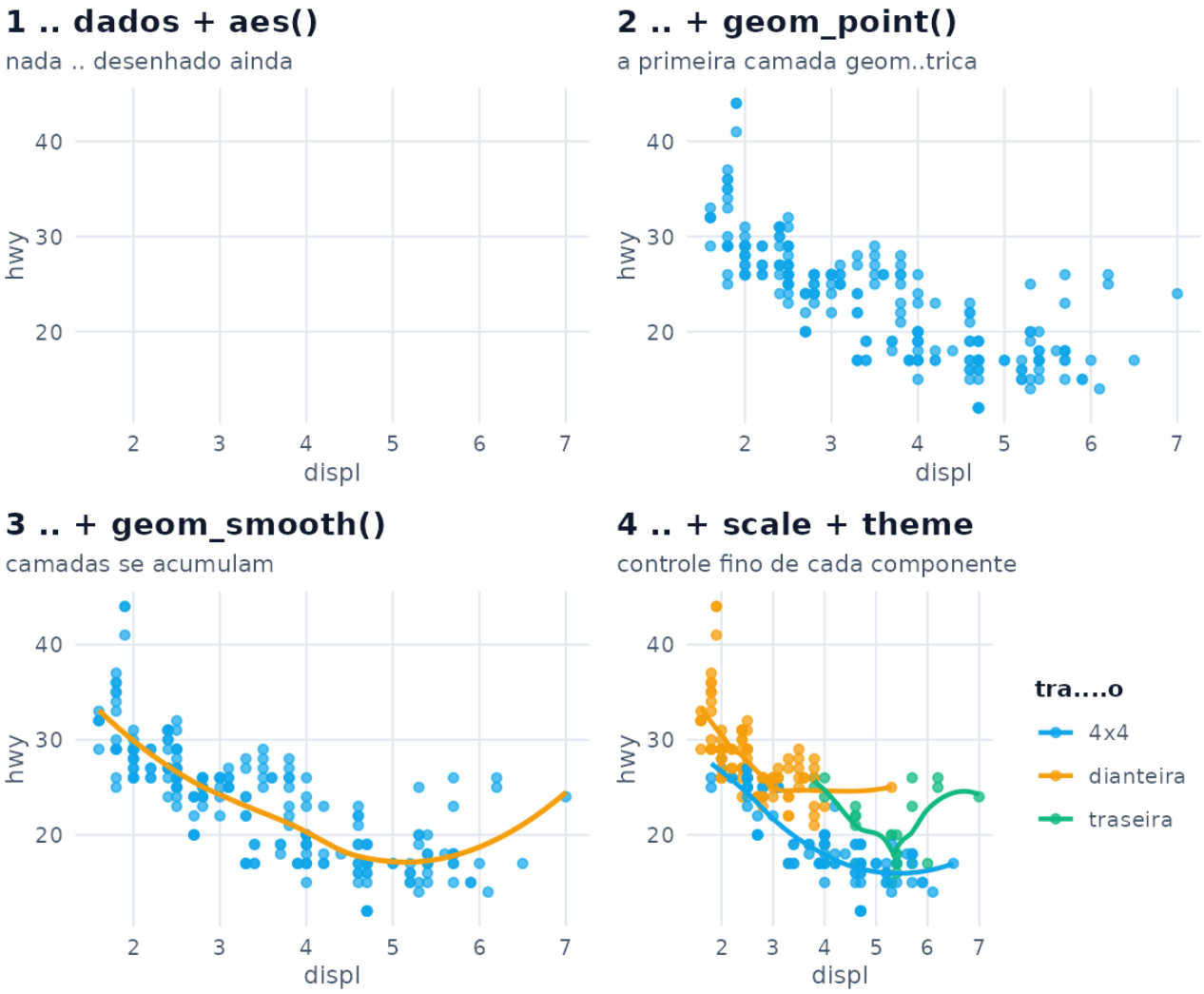


***Figura 1** — A construção em camadas: do painel vazio (dados + aes) ao gráfico controlado. Cada + adiciona um componente independente.*

CONCEITO O objeto retornado por `ggplot()` é uma **lista de especificações**, não uma imagem. O desenho só acontece no *print*. Por isso você pode guardar em variável, somar camadas depois (`p + geom_smooth()`) e construir gráficos programaticamente — a base do que faremos no Shiny.

02 Dados e aes(): o mapeamento estético

Tudo começa com dados **tidy**: cada variável é uma coluna, cada observação é uma linha, cada valor é uma célula. O ggplot2 assume esse formato — se seu gráfico está difícil de expressar, quase sempre o problema está na forma dos dados, não no gráfico (reestruture com `tidyr::pivot_longer()`).

Mapear ≠ definir

A distinção mais importante de todo o ggplot2: **dentro de `aes()`, um atributo visual codifica uma variável** (o ggplot cria escala e legenda); **fora, é uma constante** aplicada a tudo. Confundir os dois gera o bug mais clássico da biblioteca:

```r
# MAPEAR (dentro de aes): a cor CODIFICA uma variável -> escala + legenda
ggplot(mpg, aes(displ, hwy, colour = class)) + geom_point()

# DEFINIR (fora de aes): constante visual, nenhuma escala é criada
ggplot(mpg, aes(displ, hwy)) + geom_point(colour = "#0EA5E9", size = 2)

# O BUG: "azul" vira variável categórica de 1 nível -> cor padrão + legenda "azul"
ggplot(mpg, aes(displ, hwy, colour = "azul")) + geom_point()
```

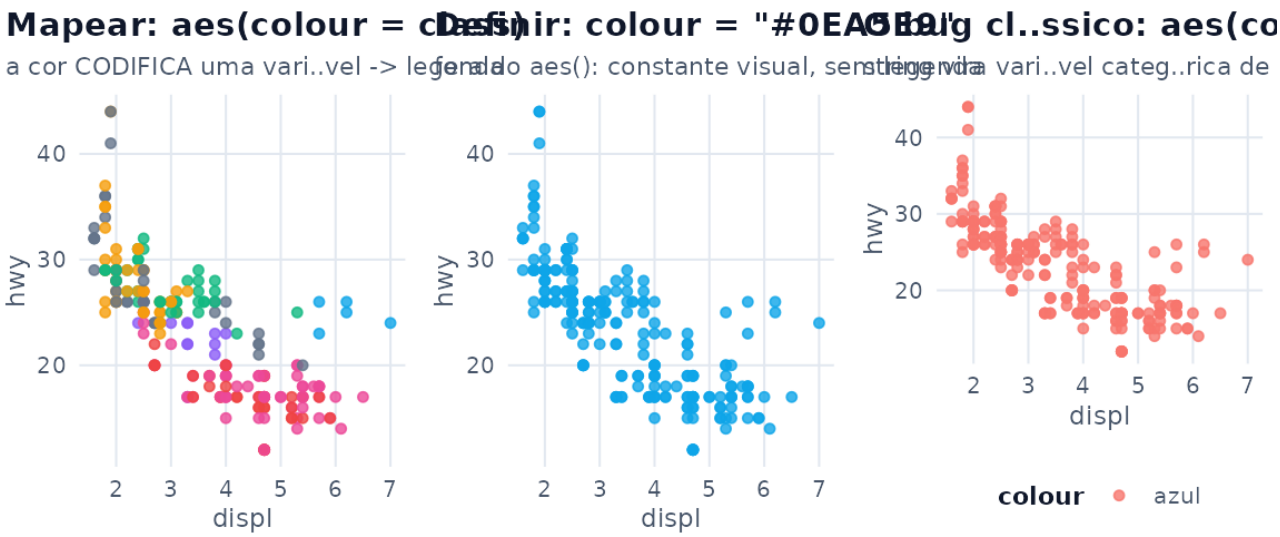


Figura 2 — Mapear cria legenda e paleta; definir aplica constante; a string dentro de aes() vira um fator de um nível — o famoso "pedi azul e veio vermelho".

Herança: aes global, aes local e dados por camada

O que vai em `ggplot()` é herdado por todas as camadas; o que vai na geom vale só para ela. Cada camada pode inclusive receber seu próprio `data` — a chave para destacar subconjuntos:

```r
# aes global (todas as camadas) vs aes local (só desta camada)
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(colour = drv)) +             # só os pontos têm cor mapeada
  geom_smooth(colour = "black", se = FALSE)    # UMA curva (sem grupos)

# Cada camada pode ter os PRÓPRIOS dados: técnica de destaque
ggplot(mpg, aes(displ, hwy)) +
  geom_point(colour = "grey75") +
  geom_point(data = subset(mpg, class == "2seater"), colour = "#EF4444", size = 3)
```

Aesthetic	Recebe bem…	Escala criada
`x, y`	contínuas, discretas, datas	`scale_x_*, scale_y_*`
`colour / fill`	discreta (paleta) ou contínua (gradiente)	`scale_colour_*, scale_fill_*`
`size`	contínua (área do ponto)	`scale_size()`
`shape`	discreta (máx. ~6 níveis legíveis)	`scale_shape()`
`alpha`	contínua (transparência)	`scale_alpha()`
`linetype`	discreta (sólida, tracejada…)	`scale_linetype()`
`group`	define agrupamento sem efeito visual	`— (não gera escala)`

DICA `colour` contorna, `fill` preenche. Pontos padrão (shape 19) só usam colour; barras, boxplots e áreas usam fill. Se a cor “não pega”, você quase certamente mapeou a aesthetic errada.

03 Geoms: as formas geométricas

A geom decide **como** as observações viram tinta: pontos, linhas, barras, áreas, intervalos. A escolha não é estética — é ditada pela pergunta e pelos tipos de variáveis envolvidas.

Pergunta	Variáveis	Geom
Relação entre duas quantidades?	2 contínuas	`geom_point(), geom_smooth()`
Evolução no tempo?	data + contínua	`geom_line(), geom_area()`
Quantos em cada categoria?	1 discreta	`geom_bar() (conta sozinha)`
Valor já calculado por categoria?	discreta + contínua	`geom_col()`
Como uma quantidade se distribui?	1 contínua	`geom_histogram(), geom_density()`
Distribuição comparada entre grupos?	discreta + contínua	`geom_boxplot(), geom_violin()`
Incerteza / intervalos?	estimativa ± erro	`geom_errorbar(), geom_ribbon()`
Rótulos e anotações?	texto posicionado	`geom_text(), geom_label(), annotate()`
Matriz de intensidade?	2 discretas + contínua	`geom_tile()`

```r
ggplot(mpg, aes(displ, hwy)) + geom_point()          # dispersão
ggplot(economics, aes(date, unemploy)) + geom_line() # série temporal
ggplot(mpg, aes(class)) + geom_bar()                 # contagem automática
ggplot(mpg, aes(class, hwy)) + geom_boxplot()        # distribuição por grupo
ggplot(mpg, aes(hwy)) + geom_histogram(bins = 20)    # distribuição
ggplot(mpg, aes(hwy, fill = drv)) + geom_density(alpha = .5)
```

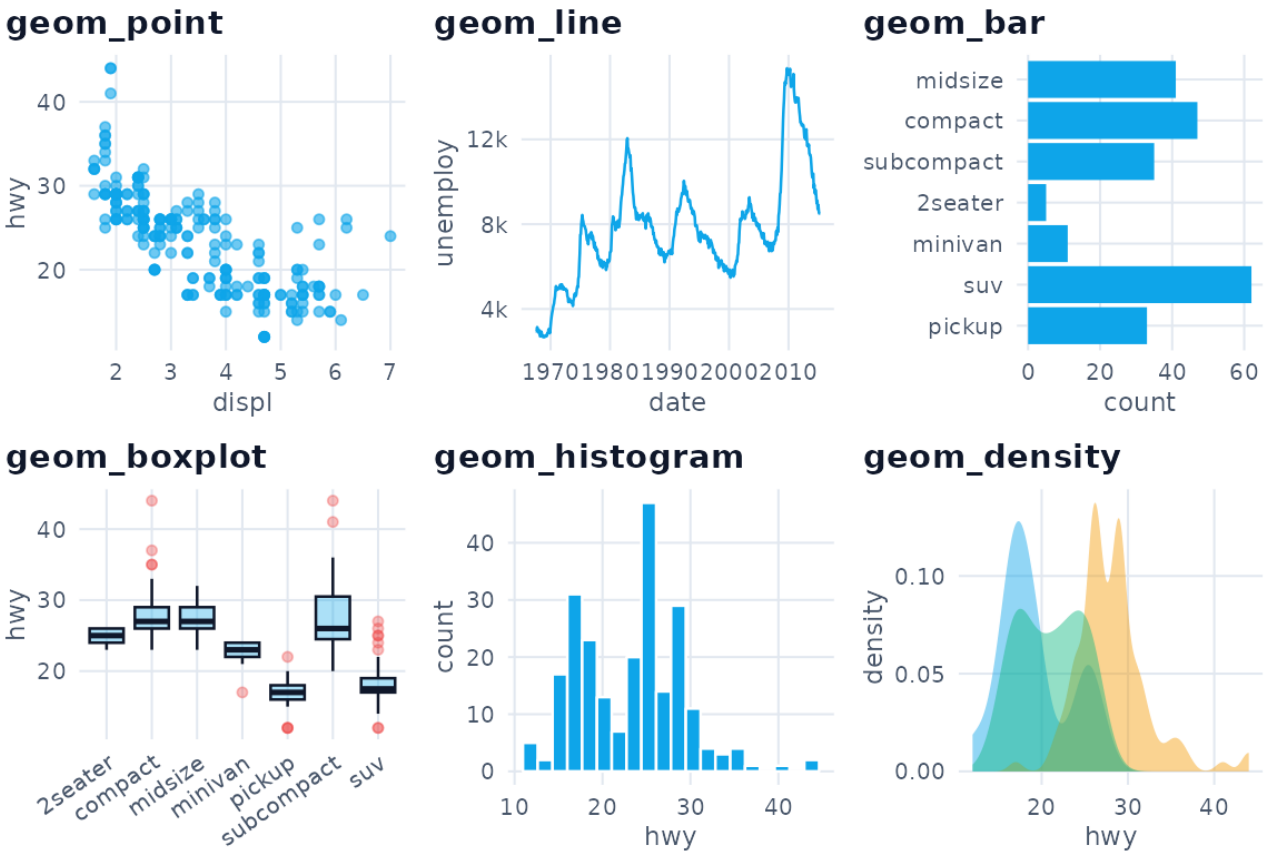


Figura 3 — Seis geoms fundamentais sobre os mesmos datasets. Note que o código difere em uma única camada — o resto da gramática é idêntico.

DICA Camadas são desenhadas **na ordem em que aparecem**: a última fica por cima. Ponha o contexto (pontos cinza, faixas de referência) primeiro e o destaque por último. Existem ~50 geoms no pacote e centenas em extensões (ggridges, ggrepel, gganimate) — todas obedecem à mesma gramática.

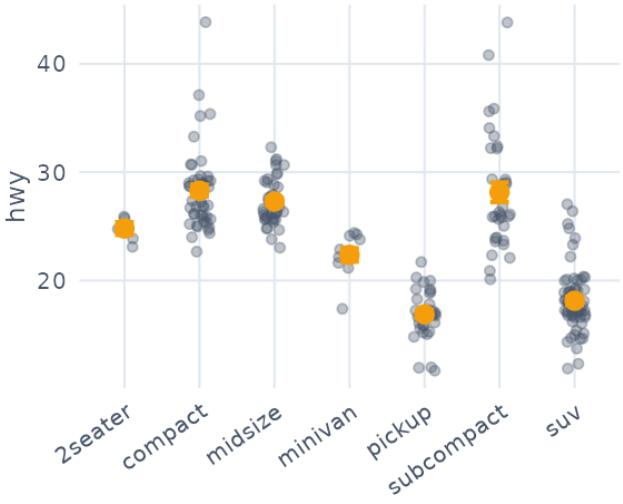
04 Stats e position: o cálculo por trás do desenho

Entre os dados e a geom existe uma transformação estatística — a **stat**. Toda geom tem uma stat padrão (e vice-versa): `geom_bar()` usa `stat_count`, `geom_histogram()` usa `stat_bin`, `geom_smooth()` usa `stat_smooth`. Quando você entende o par geom+stat, para de pré-processar dados desnecessariamente: o gráfico calcula por você.

```r
# geom_bar() é, na rigor, geom_bar(stat = "count"): a contagem é da stat
ggplot(mpg, aes(class)) + geom_bar()
ggplot(mpg, aes(class)) + stat_count()            # idêntico: geom e stat são pares

# Valor JÁ agregado na tabela? Troque a stat:
ggplot(resumo, aes(class, media)) + geom_col()    # = geom_bar(stat = "identity")

# after_stat(): acesse variáveis CALCULADAS pela stat
ggplot(mpg, aes(hwy)) +
  geom_histogram(aes(y = after_stat(density)), bins = 20)

# stat_summary(): estatística direto na camada, sem agregar antes
ggplot(mpg, aes(class, hwy)) +
  geom_jitter(width = .15, alpha = .3) +
  stat_summary(fun = mean, geom = "point", colour = "#F59E0B", size = 3) +
  stat_summary(fun.data = mean_se, geom = "errorbar", width = .25)
```

stat_summary()

m..dia .. erro-padr..o calculados pela stat



geom_smooth()

stat_smooth ajusta o modelo; geom_ribbon+line de

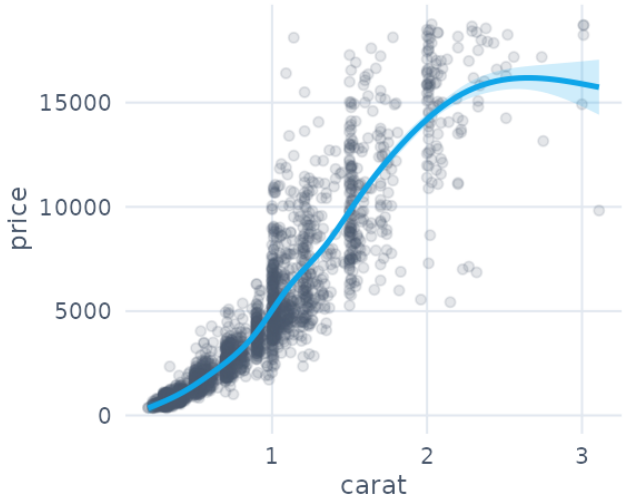


Figura 4 — Esquerda: stat_summary calcula média ± erro-padrão por grupo na hora do desenho. Direita: stat_smooth ajusta um GAM e a geom desenha linha + banda.

Position: resolvendo a sobreposição

Quando duas observações disputam o mesmo lugar, a **position** decide o desempate. É ela que transforma a mesma barra em empilhada, lado a lado ou proporcional:

```r
ggplot(mpg, aes(class, fill = drv)) + geom_bar()                    # "stack"
ggplot(mpg, aes(class, fill = drv)) + geom_bar(position = "dodge")  # lado a lado
ggplot(mpg, aes(class, fill = drv)) + geom_bar(position = "fill")   # proporção
geom_point(position = position_jitter(width = .2))                  # anti-overplot
```

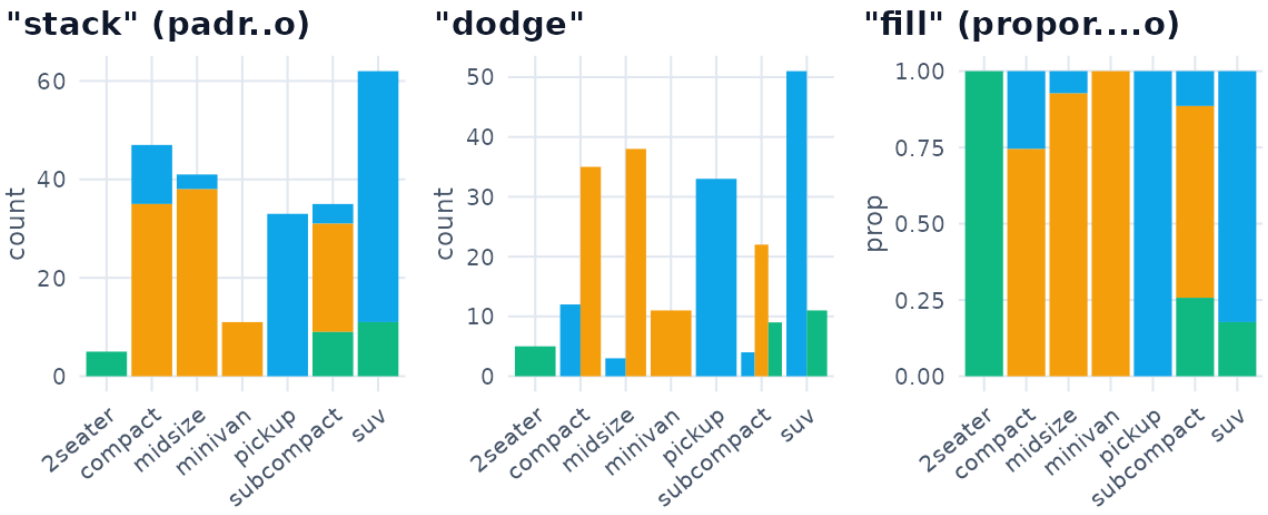


Figura 5 — O mesmo mapeamento, três positions: stack (totais), dodge (comparação dentro do grupo), fill (composição percentual).

ARMADILHA Em séries com `geom_line()`, se a linha sai em zigue-zague ou vira uma única linha ligando tudo, falta o `group` (ou uma cor mapeada) para dizer quais pontos pertencem à mesma série: `aes(group = pais)`.

05 Scales: o tradutor entre dado e percepção

Cada aesthetic mapeada ganha uma escala que traduz valores em posições, cores ou tamanhos. O nome segue uma gramática própria — `scale_<aesthetic>_<tipo>()` — e é aqui que mora a maior parte da customização fina de eixos, legendas e paletas.

```r
ggplot(diamonds, aes(carat, price, colour = price)) +
  geom_point(alpha = .4) +
  scale_x_log10() +
  scale_y_log10(labels = scales::label_dollar(prefix = "R$ ")) +
  scale_colour_viridis_c(option = "C")

# Os 4 controles universais de QUALQUER escala:
scale_y_continuous(
  limits = c(0, 50),                # domínio (CUIDADO: corta dados fora dele)
  breaks = seq(0, 50, 10),          # onde ficam as marcas
  labels = \(x) paste0(x, " mpg"),  # como as marcas são escritas
  expand = expansion(mult = .02)    # respiro nas bordas do painel
)

# Paletas discretas
scale_colour_manual(values = c("#0EA5E9", "#F59E0B", "#10B981"))
scale_colour_brewer(palette = "Dark2")     # ColorBrewer (cartografia)
scale_colour_viridis_d()                   # perceptual, daltônico-seguro

# guides(): controla a LEGENDA de cada aesthetic
guides(colour = guide_legend(nrow = 1), size = "none")
```

scale_*_log10 + viridis cont..nua transforma....o de eixo + gradiente perceptualmente uniforme

breaks, labels e limits controle total sobre eixos e paleta discreta (Brewer

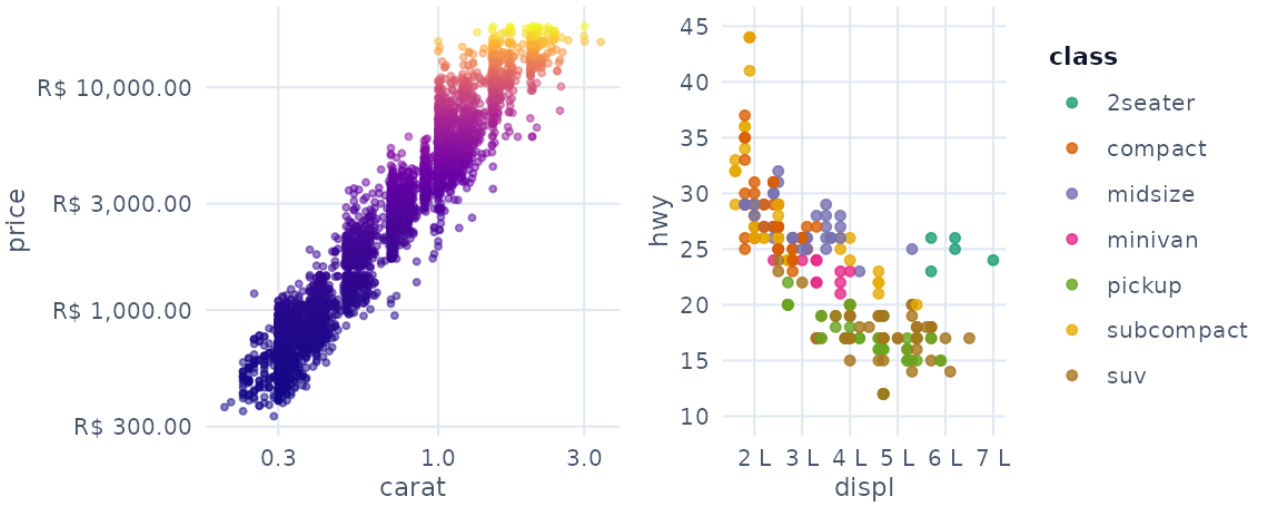


Figura 6 — Esquerda: eixos log10 + gradiente viridis contínuo (diamonds). Direita: breaks e labels customizados com paleta Brewer discreta (mpg).

Família	Quando usar	Exemplos
`*_continuous / *_log10 / *_date`	eixos numéricos, log, temporais	`scale_x_date(date_breaks = '1 year')`

Família	Quando usar	Exemplos
`*_manual`	cores exatas da sua identidade visual	`scale_fill_manual(values = pal)`
`*_brewer / *_viridis_d`	paletas discretas prontas e seguras	`scale_colour_brewer(palette = 'Set2')`
`*_viridis_c / *_gradient(2)`	gradientes contínuos	`scale_fill_gradient2(midpoint = 0)`
`labs()`	títulos de eixos, legenda, título, caption	`labs(x = NULL, colour = 'tração')`

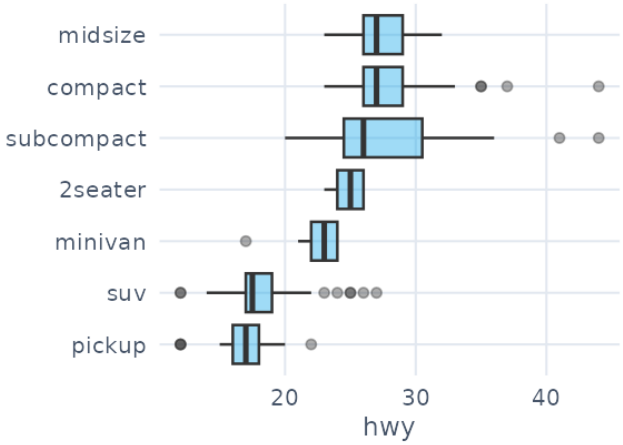
DICA O pacote `scales` resolve 90% da formatação de rótulos: `label_number()`, `label_percent()`, `label_dollar(prefix = "R$ ", big.mark = ".")`, `label_date_short()`. Nunca formate rótulo com paste() manual antes de conferir se já existe um label_*().

06 Coordenadas: a projeção final

Depois de escalas aplicadas, o sistema de coordenadas projeta as posições no plano. Trocar o coord muda a leitura do gráfico inteiro sem tocar em dados nem geoms — uma barra empilhada em coordenada polar **é** a pizza.

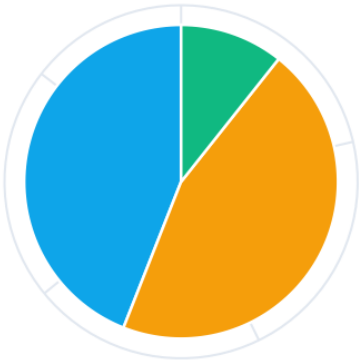
```r
p <- ggplot(mpg, aes(displ, hwy)) + geom_point() + geom_smooth()

p + coord_cartesian(ylim = c(20, 35))       # ZOOM: ajusta a "câmera", dados intactos
p + scale_y_continuous(limits = c(20, 35))  # CORTA: pontos fora viram NA e SOMEM
                                            # das stats -> o smooth muda!

p + coord_flip()                # troca x <-> y (útil p/ rótulos longos)
p + coord_fixed(ratio = 1)      # 1 unidade de x = 1 de y (mapas, PCA)

ggplot(mpg, aes(x = "", fill = drv)) +
  geom_bar(width = 1, colour = "white") +
  coord_polar(theta = "y")      # pizza = barra empilhada em polar
```

coord_flip()



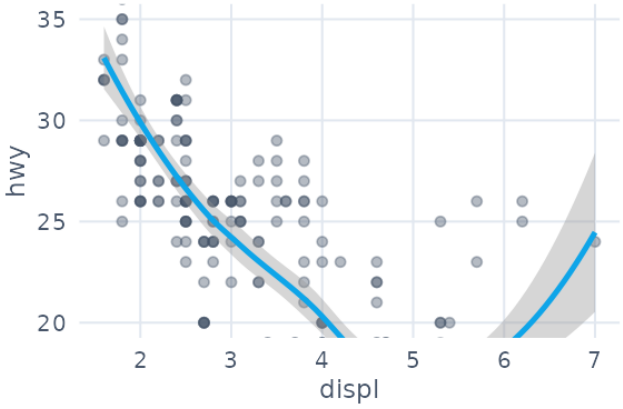
coord_polar()

barra empilhada -> pizza



coord_cartesian(ylim=...)

ZOOM: todos os dados seguem no ajuste



scale_y(limits=...)

CORTA dados: o smooth muda! (armadilha)

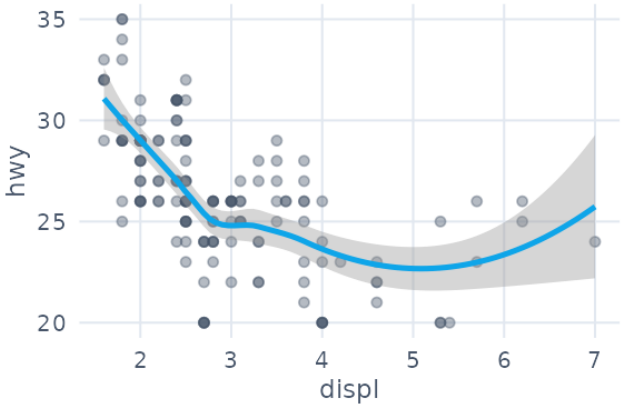


__Figura 7__ — Acima: coord_flip e coord_polar. Abaixo, a armadilha: com coord_cartesian o loess usa todos os dados (zoom honesto); com limits na escala, os pontos cortados saem do ajuste e a curva muda de forma.

ARMADILHA Para dar zoom, **sempre** `coord_cartesian(xlim=, ylim=)`. Usar `limits` na escala descarta observações antes das stats — médias, suavizações e boxplots passam a ser calculados sobre um subconjunto silenciosamente. É erro de análise, não de estilo.

07 Facetas: small multiples

Facetar é repetir o mesmo gráfico para cada subgrupo, com escalas e painéis alinhados — a forma mais honesta de comparar padrões entre categorias, superior a empilhar dez cores num painel só.

```r
# facet_wrap: UMA variável, painéis "enrolados" numa grade
ggplot(mpg, aes(displ, hwy)) + geom_point() +
  facet_wrap(~ class, nrow = 2)

# facet_grid: DUAS variáveis, linhas ~ colunas
ggplot(mpg, aes(displ, hwy)) + geom_point() +
  facet_grid(drv ~ cyl)

# Escalas livres quando as ordens de grandeza diferem entre painéis
facet_wrap(~ class, scales = "free_y")

# Truque de contexto: o TOTAL em cinza atrás de cada painel
ggplot(mpg, aes(displ, hwy)) +
  geom_point(data = transform(mpg, class = NULL), colour = "grey85") +
  geom_point(colour = "#0EA5E9") +
  facet_wrap(~ class)
```

facet_wrap(~ class)

small multiples: cada painel contra o fundo cinza do total

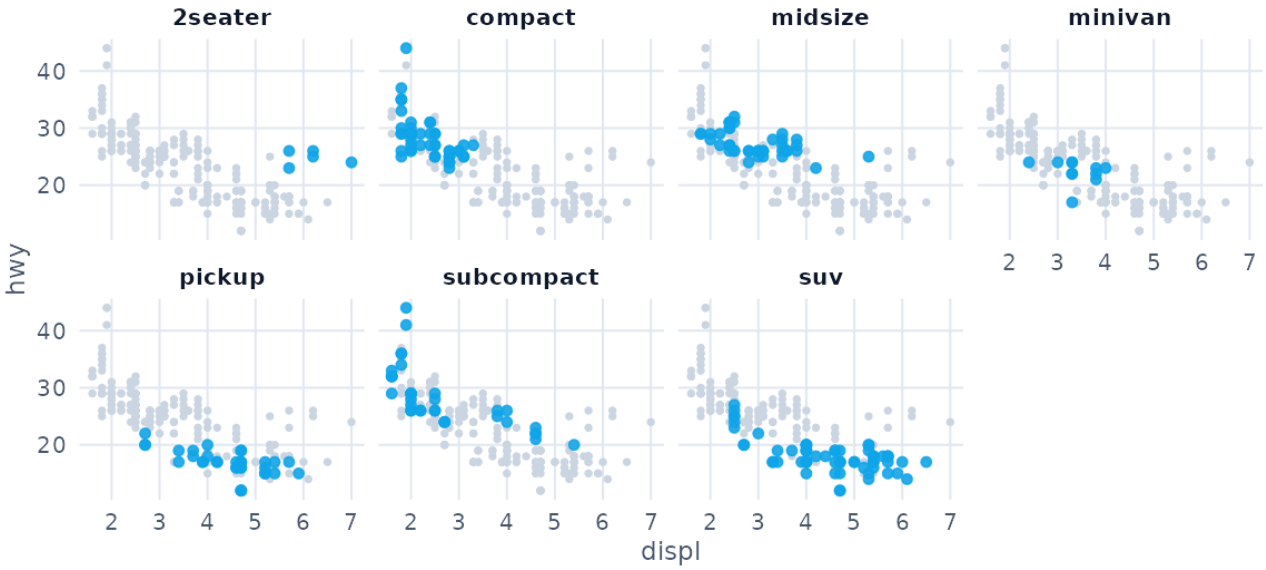


Figura 8 — facet_wrap com o truque do fundo cinza: remover a variável de facet dos dados da primeira camada faz o total aparecer em todos os painéis como referência.

DICA `scales = "free"` liberta os eixos, mas destrói a comparabilidade direta entre painéis — use apenas quando o objetivo é ver a *forma* de cada subgrupo, não comparar magnitudes.

08 Temas: customizando cada pixel não-dado

O tema controla tudo que não é dado: fontes, grades, fundos, posições de legenda, margens. A lógica é um sistema de **elementos tipados** — texto, linha, retângulo — organizados numa hierarquia com herança: `axis.title.x` herda de `axis.title`, que herda de `text`. Ajuste no nível mais alto possível e especialize só o necessário.

Elemento	Controla	Construtor
`text, title, axis.text, strip.text…`	toda tipografia	`element_text(size, face, colour, angle)`
`panel.grid.major / .minor, axis.line`	linhas e grades	`element_line(colour, linewidth, linetype)`
`panel.background, plot.background, legend.box…`	fundos e caixas	`element_rect(fill, colour)`
`qualquer um`	remover completamente	`element_blank()`
`legend.position`	posição da legenda	`"top", "bottom", "none", c(x, y)`
`plot.margin`	margens externas	`margin(t, r, b, l)`

R

```r
# Seu tema como FUNÇÃO: consistência em todos os gráficos do projeto
meu_tema <- function(base_size = 12) {
  theme_minimal(base_size = base_size) +
  theme(
    text               = element_text(colour = "#334155"),
    plot.title         = element_text(face = "bold", size = rel(1.25),
                                      colour = "#0F172A"),
    plot.subtitle      = element_text(colour = "#64748B"),
    plot.title.position = "plot",        # alinha ao gráfico, não ao painel
    panel.grid.minor   = element_blank(),
    panel.grid.major   = element_line(colour = "#E2E8F0", linewidth = .4),
    axis.ticks         = element_blank(),
    legend.position    = "top",
    strip.text         = element_text(face = "bold")
  )
}

ggplot(mpg, aes(displ, hwy)) + geom_point() + meu_tema()
theme_set(meu_tema())     # ou torne-o o padrão da sessão inteira
```

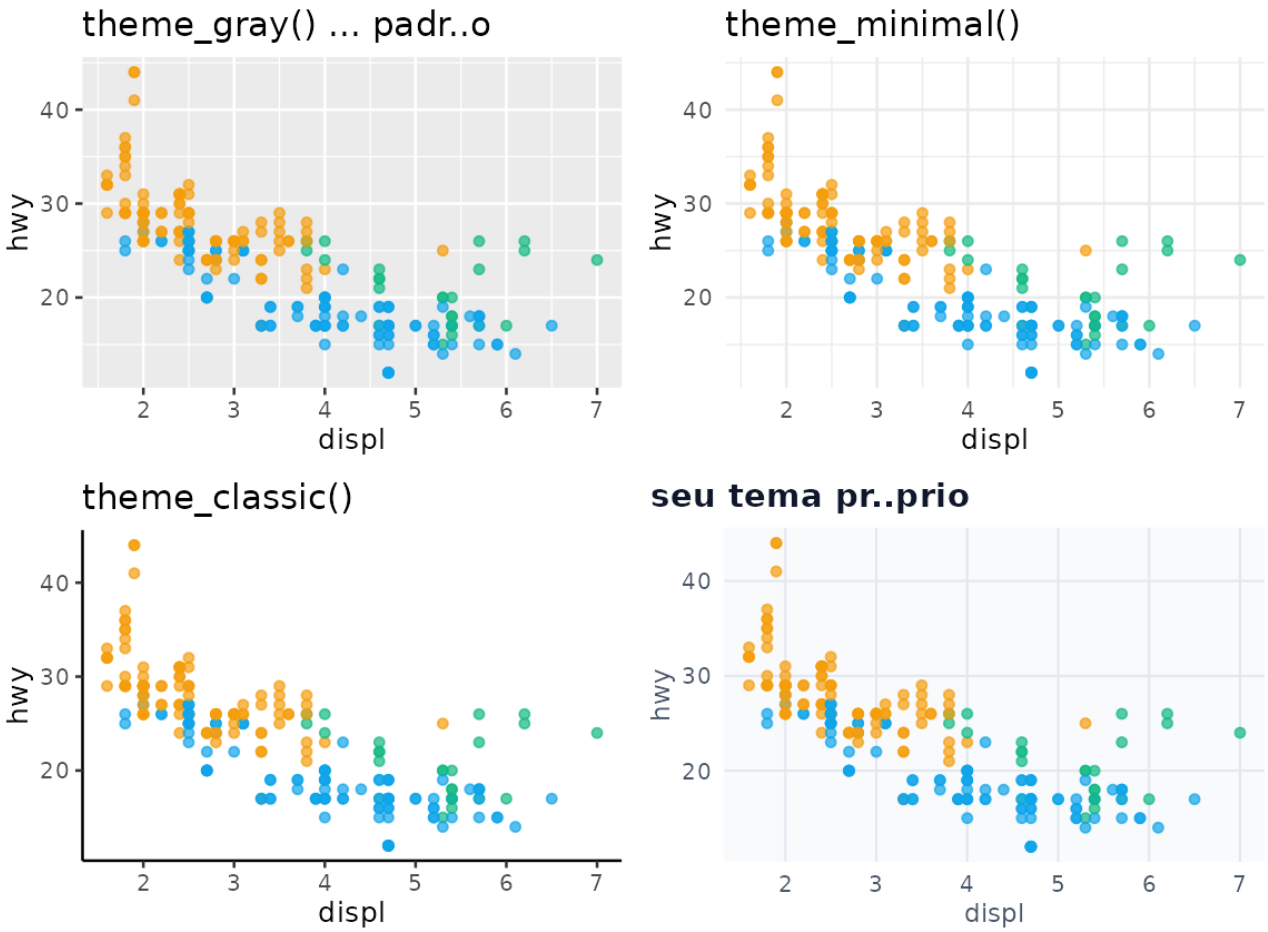


Figura 9 — Os mesmos dados sob quatro temas. Temas completos (theme_gray, theme_minimal, theme_classic) são pontos de partida; theme() sobrescreve elemento a elemento.

CONCEITO Ordem importa: `theme_minimal() + theme(...)` funciona; invertido, o tema completo **sobrescreve** seus ajustes. Tema completo primeiro, cirurgia depois.

09 Estudo de caso: do padrão ao publication-ready

Reunindo tudo: um gráfico de nível editorial usa a gramática inteira — camada de contexto, estatística na camada de destaque, reordenação de fatores, escala ajustada, coordenada invertida, título que afirma o achado (não o eixo) e tema próprio.

```r
library(ggplot2)

med <- aggregate(hwy ~ class, mpg, median)
mpg$class_o <- factor(mpg$class, levels = med$class[order(med$hwy)])  # reordena

ggplot(mpg, aes(class_o, hwy)) +
  geom_jitter(width = .12, alpha = .35, colour = "#94A3B8") +   # contexto
  stat_summary(fun = median, geom = "point",                    # destaque
               size = 4, colour = "#F59E0B") +
  stat_summary(fun = median, geom = "text",                     # rótulo da stat
               aes(label = after_stat(sprintf("%.0f", y))),
               hjust = -0.9, fontface = "bold", size = 3.2) +
  coord_flip() +                                                # rótulos legíveis
  scale_y_continuous(limits = c(10, 46), breaks = seq(10, 45, 5)) +
  labs(
    title    = "Picapes e SUVs pagam o preço na estrada",       # o ACHADO
    subtitle = "Consumo em rodovia (mpg) por categoria — ponto laranja = mediana",
    x = NULL, y = "milhas por galão (rodovia)",
    caption  = "Fonte: EPA, dataset mpg (ggplot2) · 234 modelos, 1999–2008"
  ) +
  meu_tema(12) +
  theme(panel.grid.major.y = element_blank())      # cirurgia final
```

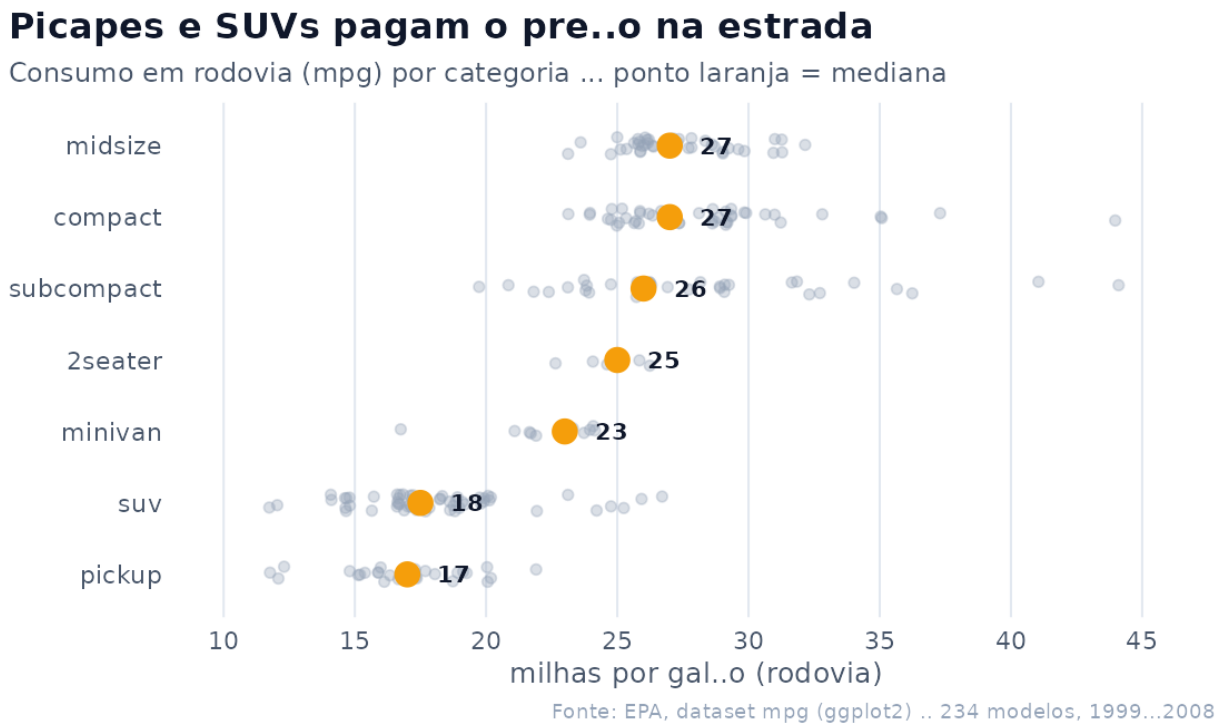


Figura 10 *— O resultado: cada decisão do código é um componente da gramática aplicado com intenção. Nada aqui exigiu função especial — só composição.*

Checklist de polimento

- **Título afirma o achado**; o subtítulo explica a métrica; o caption cita a fonte.
- **Fatores reordenados** por valor (`reorder()` / `factor(levels=)`), nunca por ordem alfabética acidental.
- **Contexto atrás, destaque na frente**: cinza neutro primeiro, cor de ênfase por último.
- **Menos tinta**: remova grades redundantes ao eixo em que já há rótulo direto.
- **Rótulos diretos** quando couberem (`geom_text`/`ggrepel`) valem mais que legenda.
- **Eixo do zero** obrigatório em barras; opcional (e declarado) em pontos e linhas.

DICA Este fluxo — EDA rápida com padrões, depois polimento por componente — é exatamente o esperado em relatórios Quarto e questões discursivas de análise de dados: justifique cada escolha pelo componente da gramática que ela troca.

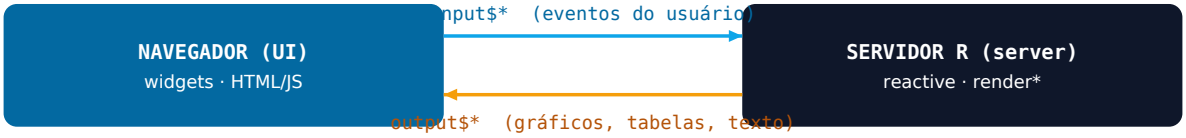
PARTE II

SHINY: DA GRAMÁTICA À WEB INTERATIVA

reatividade, inputs e o ggplot2 como motor visual

10 Arquitetura: UI, server e o contrato entre eles

Um app Shiny é um par de objetos: a **UI** declara o layout e os controles (vira HTML); o **server** é uma função que reage a inputs e produz outputs. O contrato entre os dois é nominal: o widget `selectInput("classe", …)` aparece no server como `input$classe`; o `output$disp` do server é desenhado onde a UI declarar `plotOutput("disp")`.



app.R

```
library(shiny)
library(ggplot2)

ui <- fluidPage(
  titlePanel("Explorador mpg"),
  sidebarLayout(
    sidebarPanel(
      selectInput("classe", "Categoria:", choices = unique(mpg$class)),
      sliderInput("alpha", "Transparência:", min = .1, max = 1, value = .8)
    ),
    mainPanel(plotOutput("disp"))
  )
)

server <- function(input, output, session) {
  output$disp <- renderPlot({
    dados <- subset(mpg, class == input$classe)
    ggplot(dados, aes(displ, hwy)) +
      geom_point(alpha = input$alpha, colour = "#0EA5E9", size = 3) +
      theme_minimal(base_size = 14)
  })
}

shinyApp(ui, server)   # app completo e funcional em ~25 linhas
```

No server (render)	Na UI (output)	Entrega
`renderPlot({ ggplot(...) })`	`plotOutput("id")`	gráfico (com eventos de mouse)
`renderPlotly / renderGirafe`	`plotlyOutput / girafeOutput`	ggplot com hover/zoom nativos
`renderTable / renderDT`	`tableOutput / DTOutput`	tabelas (DT: busca, paginação)

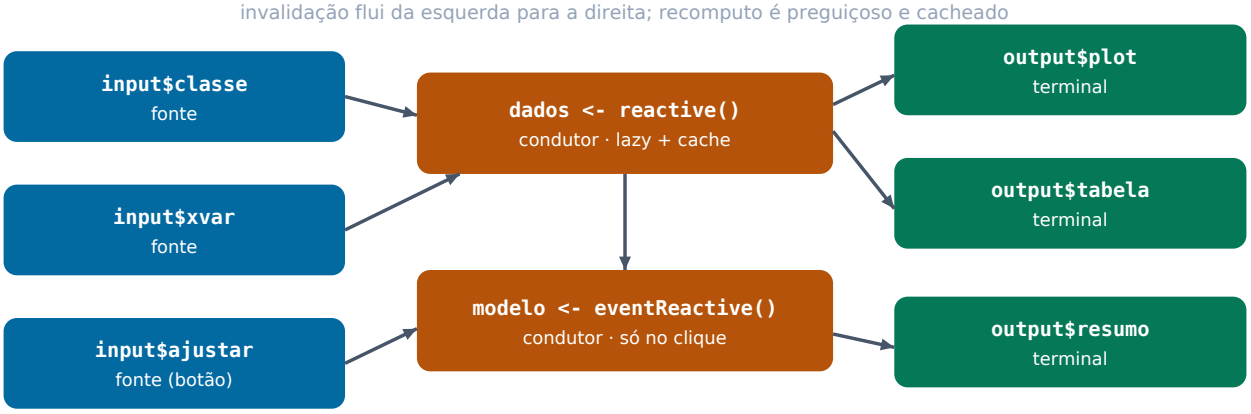
No server (render)	Na UI (output)	Entrega
`renderText / renderUI`	`textOutput / uiOutput`	texto · UI gerada dinamicamente

Widget de input	Coleta
`selectInput / selectizeInput`	escolha em lista (única ou múltipla)
`sliderInput`	número ou intervalo contínuo
`checkboxInput / radioButtons`	booleano · alternativa exclusiva
`numericInput / textInput / dateRangeInput`	número · texto livre · janela de datas
`actionButton`	gatilho de evento (pareia com eventReactive/observeEvent)

11 Reatividade a fundo: o grafo que trabalha por você

Shiny não executa o server de cima para baixo: ele constrói um **grafo de dependências** em que cada expressão reativa sabe de quais inputs depende. Quando um input muda, apenas o caminho afetado é invalidado e recalculado — de forma **preguiçosa** (só quando alguém pede o valor) e **cacheada** (um cálculo serve N consumidores).

invalidação flui da esquerda para a direita; recomputo é preguiçoso e cacheado



R

```r
server <- function(input, output, session) {

  # reactive(): expressão preguiçosa E cacheada — calcula 1x, serve N outputs
  dados <- reactive({
    req(input$classe)                    # guarda: só segue se o input existir
    subset(mpg, class == input$classe)
  })

  output$plot   <- renderPlot({ ggplot(dados(), aes(displ, hwy)) + geom_point() })
  output$tabela <- renderTable({ head(dados(), 10) })   # reusa o MESMO cálculo

  # eventReactive(): valor que SÓ recalcula quando o botão dispara
  modelo <- eventReactive(input$ajustar, {
    lm(hwy ~ displ, data = dados())
  })
  output$resumo <- renderPrint({ summary(modelo()) })

  # observeEvent(): efeito colateral (não retorna valor a ninguém)
  observeEvent(input$reset, {
    updateSliderInput(session, "alpha", value = .8)
  })
}
```

Construto	Papel no grafo	Use quando…
`reactive({...})`	condutor: valor cacheado sob demanda	o mesmo cálculo alimenta 2+ outputs
`eventReactive(gatilho, {...})`	condutor preso a um evento	cálculo caro só deve rodar no clique
`observeEvent(gatilho, {...})`	terminal de efeito colateral	atualizar widget, gravar arquivo, notificar

Construto	Papel no grafo	Use quando…
`isolate(x())`	lê sem criar dependência	usar um valor sem reagir a ele
`req(x)` / `validate(need(...))`	curto-circuito do fluxo	inputs vazios ou estados inválidos

ARMADILHA Ler `input$x` fora de contexto reativo (`render*`, `reactive`, `observe`) gera o erro clássico *"Can't access reactive value outside of reactive consumer"*. Todo acesso a input precisa estar dentro do grafo.

12 ggplot2 programável: o gráfico que o usuário monta

O passo decisivo para apps de exploração: deixar o usuário escolher **quais colunas** entram no `aes()`. O obstáculo é que `input$xvar` é uma *string* (`"displ"`), e o aes() espera um nome de coluna. A ponte moderna é o pronome `.data` do tidy eval:

```r
# ERRADO: aes(x = input$xvar)     -> mapeia a string constante, não a coluna
# CERTO:  o pronome .data indexa a coluna pelo nome guardado na string
ggplot(mpg, aes(x = .data[[input$xvar]], y = .data[[input$yvar]]))
```

E como o objeto ggplot é uma lista que só desenha no print (Cap. 1), você pode montá-lo **condicionalmente**, camada a camada, dentro do `renderPlot`. Abaixo, um explorador completo com seleção retangular (*brush*) sobre os pontos:

```r
ui <- fluidPage(
  titlePanel("Explorador de dados — ggplot2 sob demanda"),
  sidebarLayout(
    sidebarPanel(
      selectInput("x",   "Eixo X:", names(mpg), selected = "displ"),
      selectInput("y",   "Eixo Y:", names(mpg), selected = "hwy"),
      selectInput("cor", "Colorir por:", c("nenhuma", names(mpg))),
      checkboxInput("suave", "Curva de tendência", FALSE)
    ),
    mainPanel(
      plotOutput("g", brush = "sel"),   # brush: arraste um retângulo no gráfico
      h4("Pontos selecionados"),
      tableOutput("pontos")
    )
  )
)
```

◉ app.R — server

```r
server <- function(input, output, session) {

  output$g <- renderPlot({
    p <- ggplot(mpg, aes(.data[[input$x]], .data[[input$y]]))

    if (input$cor != "nenhuma")                    # camada condicional de cor
      p <- p + aes(colour = .data[[input$cor]])

    p <- p + geom_point(size = 3, alpha = .8)

    if (input$suave)                               # camada condicional de stat
      p <- p + geom_smooth(method = "loess", formula = y ~ x)

    p + theme_minimal(base_size = 15)
  }, res = 96)

  # brushedPoints casa o retângulo do usuário com as LINHAS do data.frame,
  # usando os próprios mapeamentos x/y do gráfico:
  output$pontos <- renderTable({
    brushedPoints(mpg, input$sel)
  })
}

shinyApp(ui, server)
```

Evento no plotOutput	Argumento	Leitura no server
clique simples	`click = "cl"`	`nearPoints(df, input$cl)`
duplo clique	`dblclick = "dc"`	`input$dc$x, input$dc$y (zoom!)`
retângulo arrastado	`brush = "sel"`	`brushedPoints(df, input$sel)`
hover	`hover = "hv"`	`nearPoints(df, input$hv, maxpoints = 1)`

DICA Padrão poderoso de zoom: guarde `input$sel` num `reactiveVal` no duplo clique e aplique como `coord_cartesian(xlim, ylim)` — zoom honesto (Cap. 6) controlado pelo mouse. Para hover/zoom prontos sem JavaScript, embrulhe o mesmo ggplot em `plotly::ggplotly(p)` com `renderPlotly`.

13 Performance e boas práticas

Apps lentos quase sempre violam o grafo reativo: recalculam o que já estava calculado ou reagem ao que não deviam. As correções são estruturais, não micro-otimizações:

```r
# 1. Fatore o cálculo caro num reactive — um cálculo, N consumidores
dados <- reactive({ filtra_e_agrega(base, input$filtros) })

# 2. debounce/throttle: espere o usuário terminar de digitar/arrastar
busca <- debounce(reactive(input$busca), 500)        # ms

# 3. bindCache: memoize outputs por chave — instantâneo ao revisitar
output$g <- renderPlot({ grafico_caro(input$x, input$y) }) |>
  bindCache(input$x, input$y)

# 4. Falhe cedo e com mensagem limpa
validate(need(nrow(dados()) > 0, "Nenhum dado para esses filtros."))

# 5. O que não depende de input NÃO entra no server:
#    leia arquivos e ajuste temas no topo do app.R (roda 1x por processo)
```

- **Plote menos**: acima de ~50 mil pontos, agregue (`geom_hex`, `geom_bin2d`) ou amostre — o gargalo é o desenho, não o R.
- `theme_set(meu_tema())` no topo do app: consistência visual sem repetir tema em cada renderPlot.
- Prefira `reactive` (valor) a `observe` + variável global — efeitos colaterais escondem dependências do grafo.
- Meça antes de otimizar: `profvis::profvis(runApp())` mostra onde o tempo realmente vai.
- Publique com `shinylive` (roda no navegador, sem servidor) ou shinyapps.io / Posit Connect.

14 Cheat sheet: consulta rápida

A gramática em uma página — o mapa mental para compor qualquer gráfico e levá-lo à web.

Quero… → use

Quero…	Use
destacar um subgrupo	`2ª camada geom_point(data = subset(...), colour = destaque)`
barras ordenadas por valor	`aes(x = reorder(cat, valor)) ou forcats::fct_reorder()`
eixo em log	`scale_x_log10() · scale_y_log10()`
porcentagem no eixo	`scale_y_continuous(labels = scales::label_percent())`
zoom sem viés	`coord_cartesian(xlim =, ylim =) — nunca limits na escala`
girar rótulos do eixo x	`theme(axis.text.x = element_text(angle = 45, hjust = 1))`
legenda no topo / removida	`theme(legend.position = "top") · "none"`
remover um título de eixo	`labs(x = NULL)`
anotar um ponto específico	`annotate("text", x =, y =, label =) — fora dos dados`
salvar em alta qualidade	`ggsave("fig.png", width = 8, height = 5, dpi = 300)`
média ± erro por grupo	`stat_summary(fun.data = mean_se, geom = "errorbar")`
muitos pontos sobrepostos	`alpha, position_jitter(), geom_hex(), geom_bin2d()`

Shiny essencial

Padrão	Receita
input controla coluna do aes	`aes(.data[[input$var]])`
um cálculo, vários outputs	`d <- reactive({...}); use d() em cada render*`
rodar só no clique	`eventReactive(input$botao, {...})`
efeito colateral (update, arquivo)	`observeEvent(input$x, {...})`
seleção de pontos no gráfico	`plotOutput(brush=) + brushedPoints(df, input$sel)`
app instantâneo ao revisitar	`renderPlot({...}) |> bindCache(inputs...)`
não quebrar com input vazio	`req(input$x) · validate(need(cond, msg))`

CONCEITO A síntese de todo o guia: **gráfico = dados + mapeamentos + camadas componíveis**; **app = grafo reativo que reconstrói esse gráfico a cada mudança de input**. Dominar os dois lados é dominar a mesma ideia — declarar *o quê*, e deixar o motor decidir *quando* e *como*.

Referências: Wickham, H. *ggplot2: Elegant Graphics for Data Analysis* (3ª ed., ggplot2-book.org) · Wickham, H. *Mastering Shiny* (mastering-shiny.org) · Wilkinson, L. *The Grammar of Graphics* (2005) · Documentação: ggplot2.tidyverse.org, shiny.posit.co.